



# Embedded Functions for Constraints and Variable Strength in Combinatorial Testing

George B. Sherwood

Open <http://testcover.com/pub/background/iwct2016.pptx>  
with PowerPoint for presentation notes and animations





# Project objectives and this talk

- Testcover.com service usability enhancement project
  - Simple functions to describe constraints
  - Automatic function evaluation and test case generation
  - Variable strength designs
  - Suitable response times
  - Practical numbers of test cases
- Topics
  - **Calendar constraints:** Conform to constraints with a simple function
  - **Body Mass Index (BMI) report classes:** Use equivalence class functions to reach expected results classes
  - **Shopping cart states:** Simplify complex behavior with functions
  - **Variable strength:** Use hybrid functions for higher strength subarrays while conforming to other constraints





# Test model terms

- There are  $k$  test factors, e.g. configurations & inputs
- A test case has 1 value for each factor (i.e. a  $k$ -tuple)
- A strength- $t$  design ( $t \leq k$ ) includes all required  $t$ -tuples of test factor values
- A *partition* includes the allowed combinations ( $t$ -tuples) for 1 test case generation instance
- An *equivalence class* includes combinations for 1 class of expected results
- A *constraint* is a condition to specify allowed factor values depending on the values of other factors
- *Masking* occurs when required tuples are missing from test cases for a class of results





# Functionally dependent test factor values

- Constraints can be described using functionally dependent test factor values
- Functional dependence:
  - 1 or more values of a dependent factor are identified by other, determinant factors
  - Determinant factors' values  $\rightarrow$  dependent factor values
- Example: The last day of any month is identified its month and year
  - Month, Year  $\rightarrow$  Last day values
  - $\ell$  = number of determinant factors ( $\ell = 2$  in this example)
- Use Direct Product Block (DPB) notation with or without embedded combination functions



# Direct Product Block (DPB) notation

## Fixed values form

Calendar Example without last\_day function  
Month  
Day  
Year  
#ok All good dates  
jan feb mar apr may jun jul aug sep oct nov dec  
1 10  
2015 2016 2017  
+ long month last day  
jan mar may jul aug oct dec  
31  
2015 2016 2017  
+ short month last day  
apr jun sep nov  
30  
2015 2016 2017  
+ feb last day  
feb  
28  
2015 2017  
+ leap day  
feb  
29  
2016

- Valid calendar dates example with boundary checking
- Factor values are on separate lines
- All combinations in a block are allowed
- Partition of multiple blocks includes union of their allowed combinations



# Direct Product Block (DPB) notation

## Fixed values form

Calendar Example without last\_day function  
Month  
Day  
Year  
#ok All good dates  
jan feb mar apr may jun jul aug sep oct nov dec  
1 10  
2015 2016 2017  
+ long month last day  
jan mar may jul aug oct dec  
31  
2015 2016 2017  
+ short month last day  
apr jun sep nov  
30  
2015 2016 2017  
+ feb last day  
feb  
28  
2015 2017  
+ leap day  
feb  
29  
2016

## Functionally dependent form

Calendar Example with last\_day function  
\$month  
Day  
\$year  
#ok All good dates  
jan feb mar apr may jun jul aug sep oct nov dec  
1 10 last\_day(\$month,\$year)  
2015 2016 2017

- Month, Year → Last day values
- Factors renamed as variables for function arguments:  
\$month \$year
- Day values:  
1 10 last\_day(\$month,\$year)
- 5 blocks now represented by only 1 block



# Evaluation of calendar last\_day function

## Fixed values

+ Day: 1 10  
.....  
.....  
.....

+ Day: 31  
.....  
.....  
.....

+ Day: 30  
.....  
.....  
.....

+ Day: 28  
.....  
.....  
.....

+ Day: 29  
.....  
.....  
.....

## 5 FV blocks

Test cases  
1 .....  
2 .....  
3 .....  
.....  
.....

## Functionally dependent

+ FD block  
.....  
.....  $cf_1$  .....  
.....

+ Day: 1 10  
.....  
.....  
.....

+ Day: 31  
.....  
.....  
.....

+ Day: 30  
.....  
.....  
.....

+ Day: 28  
.....  
.....  
.....

+ Day: 29  
.....  
.....  
.....

## 5 FV blocks

Test cases  
1 .....  
2 .....  
3 .....  
.....  
.....

All fixed values 1 10 last\_day(\$month,\$year)

FV: 1 10 Evaluation & collection

$cf_1$ : last\_day(\$month,\$year)





# Equivalence class functions

- Equivalence class functions identify classes of expected results
- `Adult_report($Age,$Weight,$Height)` returns report class:  
underweight   normal   overweight   obese   no
- Combinations of determinant values are required
- An equivalence class factor has all its class values  
because all of its determinant combinations are evaluated
- Pairwise test case generation covers all of these classes  
because they are values of the equivalence class factor
- Equivalence classes are paired with nondeterminant factors
- Higher strength not needed to reach well defined classes



# Equivalence class factors

## Functionally dependent form

Body Mass Index Report Application - 1 partition  
with EC factors

\$Age

\$Weight

\$Height

\$Sex

Intake

Medicare equivalence class

Child equivalence class

Adult equivalence class

#

19 42 67

131 180

64 71

female male

2000 3000

Medicare\_report(\$Age)

Child\_report(\$Age,\$Sex)

Adult\_report(\$Age,\$Weight,\$Height)

- 5 input factors
- 3 equivalence class factors

Medicare\_report

\$Age  $\geq$  65: yes

\$Age < 65: no

Child\_report

\$Age  $\geq$  20: no

\$Sex = female: girl

\$Sex = male: boy

Adult\_report

\$Age < 20: no

else: BMI classification



# Evaluation of equivalence class functions

+ FD block  
 ...  $cf_1$  ...  
 ...  $cf_2$  ...  
 ...  $cf_3$  ...

**\$Age: 19 42 67**

+ Medicare: yes  
 ...  
 ...  $cf_2$  ...  
 ...  $cf_3$  ...

**\$Age: 67**

+ Child: no  
 ...  
 ...  $cf_3$  ...

**\$Age: 67**

+ Adult: underweight

+ Adult: normal

+ Adult: overweight

+ Adult: obese

...

...

+ Adult: underweight

+ Adult: normal

+ Adult: overweight

+ Adult: obese

...

...

+ Adult: no

...

...

...

...

+ Adult: no

...

...

...

+ Medicare: no  
 ...  
 ...  $cf_2$  ...  
 ...  $cf_3$  ...

**\$Age: 19 42**

+ Child: no  
 ...  
 ...  $cf_3$  ...

**\$Age: 42**

+ Child: boy  
 ...  
 ...  $cf_3$  ...

**\$Age: 19**

+ Child: girl  
 ...  
 ...  $cf_3$  ...

**\$Age: 19**

10 FV blocks

Test cases  
 1 ...  
 2 ...  
 3 ...  
 ...  
 ...  
 ...

Evaluation & collection

$cf_1$ : Medicare\_report(\$Age)

$cf_2$ : Child\_report(\$Age,\$Sex)

$cf_3$ : Adult\_report(\$Age,\$Weight,\$Height)



# Constraint simplification

## Instant Shopping

Your cart contains:

Delete	Item Number	Item Description	Quantity	Price	Item Total
<input type="checkbox"/>	itemA	descriptionA	<input type="text" value="1"/>	14.95	14.95
<input type="checkbox"/>	itemB	descriptionB	<input type="text" value="2"/>	9.95	19.90
<input type="checkbox"/>	itemC	descriptionC	<input type="text" value="1"/>	5.95	5.95
					+ _____
<b>Subtotal:</b>					\$ 40.80

< Shop

Update

Checkout >

### Constraints:

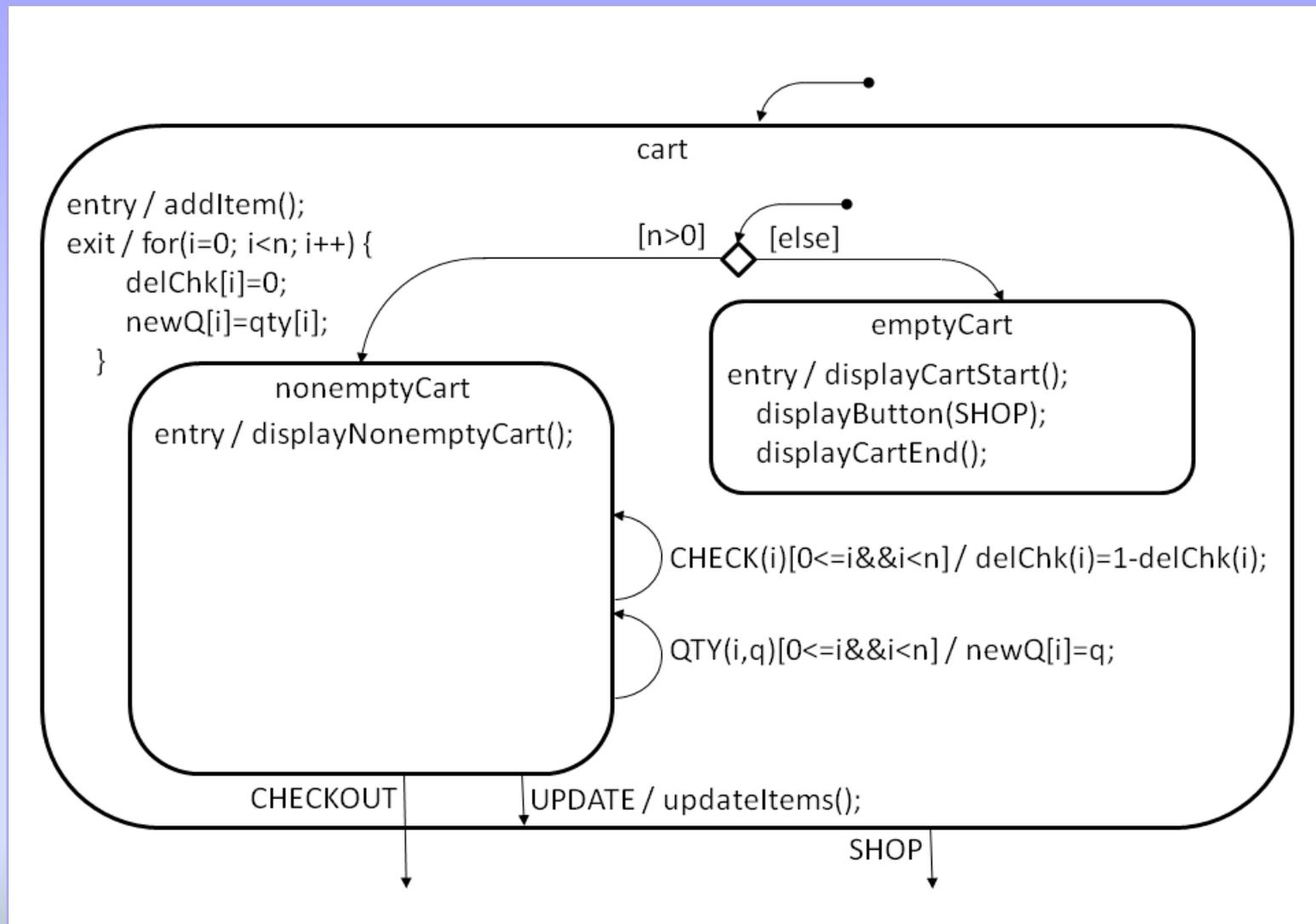
- Items in different positions must be different
- Factor values may be NULL (unused)
- Equivalence classes are target UML leaf states to avoid masking

### Simplification for nonemptyCart to nonemptyCart transition:

- Without combination functions: 33 blocks
- With 9 combination functions: 3 blocks
- Average function length: 7 lines



# Shopping cart state diagram



# Shopping cart test factors

- Program variables
  - Indexed for cart position
  - \$delChk[0]
  - \$item[0]
  - \$qty[0]
  - \$newQ[0]
- Current state
  - nonemptyCart
- Event (trigger)
  - CHECK QTY UPDATE

Test Factor	Test Factor Values	Combination Functions	Indication
\$newItem	NULL		Item to place in cart
\$n	1 2 3		Number of items in cart
\$delChk[0]	0 1	f_delChk_CQ f_delChk_U	Delete box checked in cart position 0
\$item[0]	itemA itemB itemC	f_item	Item in cart position 0
\$qty[0]	1 2 10	f_qty	Quantity of item in cart position 0
\$newQ[0]	0 1 2 10	f_newQ_CQ f_newQ_U	New quantity shown in cart position 0
\$delChk[1]	0 1 NULL	f_delChk_CQ f_delChk_U	Delete box checked in cart position 1
\$item[1]	itemA itemB itemC NULL	f_item	Item in cart position 1
\$qty[1]	1 2 10 NULL	f_qty	Quantity of item in cart position 1
\$newQ[1]	0 1 2 10 NULL	f_newQ_CQ f_newQ_U	New quantity shown in cart position 1
\$delChk[2]	0 1 NULL	f_delChk_CQ f_delChk_U	Delete box checked in cart position 2
\$item[2]	itemA itemB itemC NULL	f_item	Item in cart position 2
\$qty[2]	1 2 10 NULL	f_qty	Quantity of item in cart position 2
\$newQ[2]	0 1 2 10 NULL	f_newQ_CQ f_newQ_U	New quantity shown in cart position 2
\$i	0 1 2 NULL	f_i	Cart position for event
\$q	0 1 2 10 NULL		Quantity for event
state	nonemptyCart		Source state
event	CHECK(\$i) QTY(\$i,\$q) UPDATE	f_event_CHECK f_event_QTY	Trigger to target state



# Shopping cart blocks

- DPB request in FD form
- 18 factor names
- Values in 3 blocks
- Allowed item values:  
itemA itemB itemC  
NULL
- f\_item function returns  
item values in all 3  
blocks for:  
\$item[0]  
\$item[1]  
\$item[2]

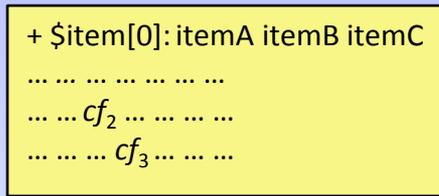
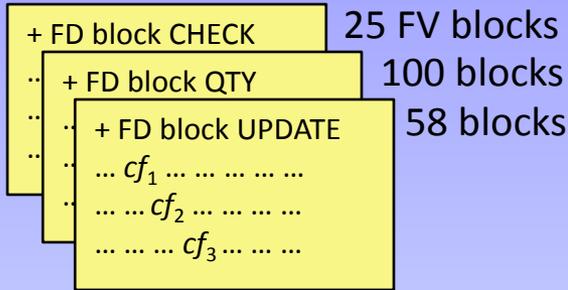
```
Shopping Cart - transition design - SCA (2,18)
$NewItem
$N
$delChk[0]
$item[0]
$qty[0]
$newQ[0]
$delChk[1]
$item[1]
$qty[1]
$newQ[1]
$delChk[2]
$item[2]
$qty[2]
$newQ[2]
$i
$q
State
Event
#NN nonemptyCart to nonemptyCart
+ nonemptyCart to nonemptyCart; CHECK
NULL
1 2 3
f_delChk_CQ(0,$n)
f_item(0,$n,,)
f_qty(0,$n)
f_newQ_CQ(0,$n)
f_delChk_CQ(1,$n)
f_item(1,$n,$item[0],)
f_qty(1,$n)
f_newQ_CQ(1,$n)
f_delChk_CQ(2,$n)
f_item(2,$n,$item[0],$item[1])
f_qty(2,$n)
f_newQ_CQ(2,$n)
f_i($n)
NULL
nonemptyCart
f_event_CHECK($i)
```

```
+ nonemptyCart to nonemptyCart; QTY
NULL
1 2 3
f_delChk_CQ(0,$n)
f_item(0,$n,,)
f_qty(0,$n)
f_newQ_CQ(0,$n)
f_delChk_CQ(1,$n)
f_item(1,$n,$item[0],)
f_qty(1,$n)
f_newQ_CQ(1,$n)
f_delChk_CQ(2,$n)
f_item(2,$n,$item[0],$item[1])
f_qty(2,$n)
f_newQ_CQ(2,$n)
f_i($n)
0 1 2 10
nonemptyCart
f_event_QTY($i,$q)

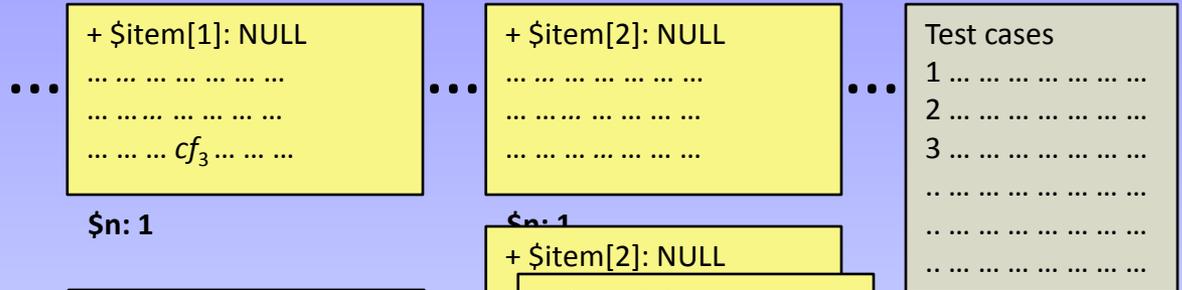
+ nonemptyCart to nonemptyCart; UPDATE
NULL
1 2 3
f_delChk_U(0,$n,1,0,1,0)
f_item(0,$n,,)
f_qty(0,$n)
f_newQ_U(0,$n,1,0,1,0)
f_delChk_U(1,$n,$delChk[0],$newQ[0],1,0)
f_item(1,$n,$item[0],)
f_qty(1,$n)
f_newQ_U(1,$n,$delChk[0],$newQ[0],1,0)
f_delChk_U(2,$n,$delChk[0],$newQ[0],$delChk[1],$newQ[1])
f_item(2,$n,$item[0],$item[1])
f_qty(2,$n)
f_newQ_U(2,$n,$delChk[0],$newQ[0],$delChk[1],$newQ[1])
NULL
NULL
nonemptyCart
UPDATE
```



# Evaluation of composite functions



\$n: 1 2 3

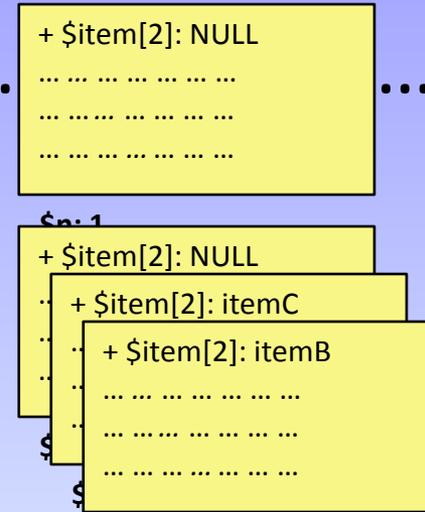


\$n: 1

\$n: 2 3

\$n: 2 3

\$n: 2 3

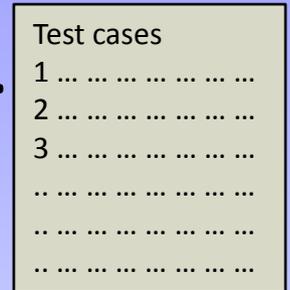


\$n: 1

\$n: 2

\$n: 2

\$n: 2



Test cases

- 1 .....
- 2 .....
- 3 .....
- .....
- .....
- .....
- .....
- .....

Evaluation & collection

- $cf_1: f\_item(0, \$n, ,)$
- $cf_2: f\_item(1, \$n, \$item[0],)$
- $cf_3: f\_item(2, \$n, \$item[0], \$item[1])$



# Hybrid functions for variable strength

- Hybrid functions return tuples of their arguments  
pair(\$a,\$b) returns (\$a,\$b)      triple(\$a,\$b,\$c) returns (\$a,\$b,\$c)
- Hybrid factors force higher strength in pairwise designs  
Factors (\$a,\$b) & (\$c,\$d) force strength 4 coverage of factors \$a \$b \$c \$d
- Hybrid factors were defined for variable strengths  $\leq 6$
- Hybrid functions inherit values from their superblocks  
So they conform to other test model constraints if present
- Variable strength test cases were generated for the shopping cart example, e.g. 2 triple factors were used for strength 6:  
(\$delChk[0],\$newQ[0],\$delChk[1])      (\$newQ[1],\$delChk[2],\$newQ[2])





# Conclusions

- Test designs can conform to constraints described by simple functions in a familiar language
- Embedded functions results are equivalent to using fixed values
- Equivalence class functions can eliminate the need for higher strength generation to reach classes of expected results
- Embedded functions can simplify testing complex systems through encapsulation and composite relations
- Hybrid functions can increase strength among selected test factors while conforming to system constraints



# Next steps

- Substitution function implementation
- Interface to include and manage functions
- Additional resource management
- Security considerations
- More testing

